# 加餐｜基于纯浏览器的视频会议方案探究

2021-12-24 章韦

《攻克视频技术》 课程介绍 >

**讲述：山荣**

时长 10:57 大小 10.04M

你好，我是章韦。目前就职于某知名外企，任职 Technical Leader，主要研究方向为信令控制及媒体传输相关领域。很开心受邀来到李江老师的专栏做一期加餐分享。今天我会围绕“基于纯浏览器的视频会议方案”谈一谈我的想法，期待能给你带来一些启发。

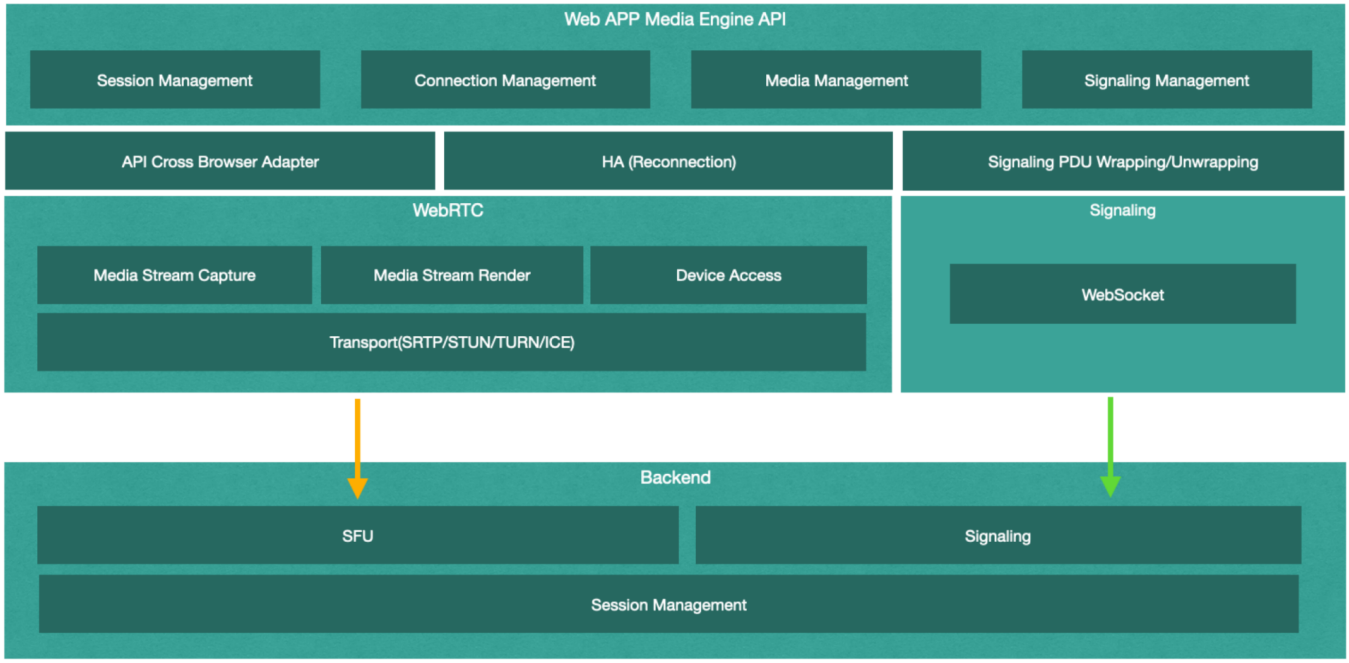
随着视频编解码技术的快速发展以及带宽成本的逐年降低，视频技术被广泛应用于生产生活中各个不同的领域，而视频会议则是其中最广泛的应用之一。伴随着这几年疫情在全球肆虐，各式各样的商业视频会议软件如雨后春笋，从全球范围来说，除了老牌的 Cisco Webex、微软的 Teams，还有新晋王者 Zoom Meetings， 国内来说，不论是阿里还是腾讯，都推出了自己的视频会议软件。

那么这节课我们就偏居一隅，聊一聊各大视频会议厂商在基于纯浏览器视频会议上的一些尝试和遇到的问题，以及各主流浏览器在底层接口上提供的一些支持，并对其中涉及到的

一些技术栈展开一定介绍。

## WebRTC

提到 Web App 和浏览器对媒体传输的支持，首先不得说 WebRTC，典型的基于 WebRTC 的会议系统架构如下：




结合上面的架构图和流程图来看，WebRTC 的 Web App 似乎并不复杂，几乎所有的前后端模块都能找到成熟的开源实现。**然而对于传统的视频会议提供商来说，支持 WebRTC 却并非这么简单。**

首先要考虑到的问题就是 **QoS**。对于成熟的视频会议企业来说，无论是 ARQ、FEC 还是 CC，都会有一套自己私有的网络适应性算法（QoS）框架， 但这些经过长时间验证且成熟的网络适应性算法却无法直接用在 Web App 后端的 SFU，因为浏览器端并不支持。

更进一步来说，不同的浏览器，其自身的**网络适应性兼容方案**也并不一致，再考虑到各种浏览器版本升级带来的进一步差异，在服务端做兼容各浏览器的网络适应性算法（QoS）其成本显然是相当高的。仅拿拥塞控制（Congest Control）举例来说，至少要分别实现 GCC/NADA/SCReAM 算法和 REMB/Transport-CC 报文来兼容不同的浏览器。

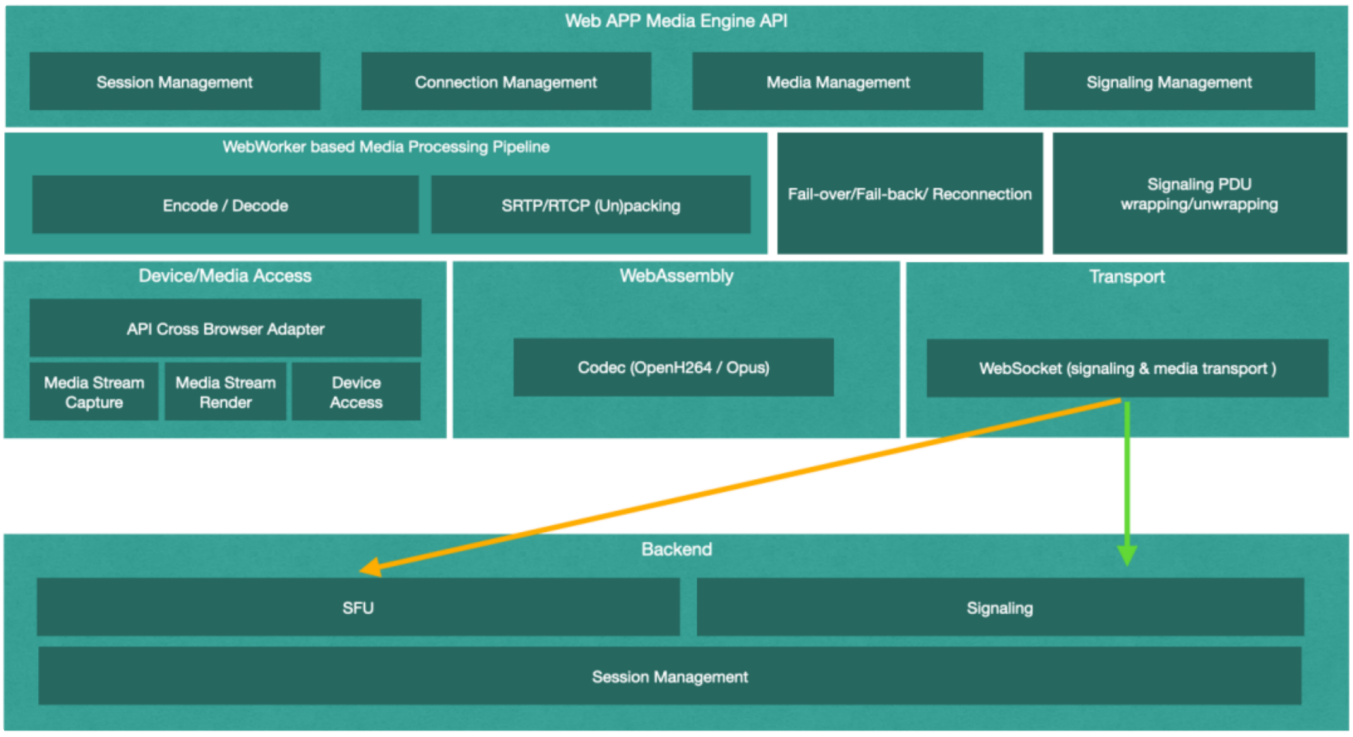
除了服务端的 QoS 问题之外，客户端部分也会有较大的问题，因为 WebRTC 诞生之初想解决的仅仅是 P2P 的媒体传输而并非会议，它提供了基于 P2P 传输简单易用的一揽子方案，但却也因此**没有提供更多底层媒体处理接口**。这就意味着开发者没有办法进一步处理

任何的媒体流，进而无法在 Web App 中提供很多富有竞争力的功能，比如各种实时滤镜、虚拟背景、端对端加密等等。虽然 Chrome 开始提供 MediaStreamTrack 的 Insertable Stream 接口可以处理 WebRTC 中的媒体流，但要等到各主流浏览器都开始支持，显然还是路漫漫其修远兮！

除此之外，另一个更麻烦的问题是 **WebRTC 中的 TURN/STUN 对于 TCP 443 端口的兼容性问题**。很多企业内部的网络防火墙仅对外开放 TCP 443 端口，且该端口仅允许 TLS 数据包通过，但所有浏览器 WebRTC 的 TURN/STUN 实现都不支持 TLS，这意味我们需要为这种情况做失败转移。而遗憾的是，这里并没有简单的失败转移方案。

## 失败转移方案

上面我们提到，Web App 为了支持仅开放了 443 端口的企业，不得不实现一套新的失败转移方案。而这套方案需要彻头彻尾的重新实现，其大致架构如下：




从上图我们可以看出，这里我们不再将媒体的处理和传输部分托管给 WebRTC，取而代之的是 一个基于 WebWorker 的媒体流水线处理机（WebWorker based Media Processing Pipeline），用来对从本地设备采集到的设备进行编码，并打包成 RTP 包，再通过 WebSocket 信令通道发送给服务端。

**完整的视频流发送过程是这样的。**

首先通过 getUserMedia（getDisplayMedia）接口从设备获取视频流，再每间隔 33 毫秒（帧率为 30）执行以下流程：

1. 将此视频流当前帧通过 Canvas API 绘制到画布中；
2. 通过 Canvas API 读取画布中当前帧的 RGBA 图像；
3. 将 RGBA 图像数据转化为 YUV 格式；
4. 将 YUV 格式的图像数据发送到 OpenH264 模块进行编码；
5. 将编码后的数据包根据不同的 NALUnitType 封装为不同的 RTP Payload；
6. 将封装后的 RTP Payload 打包为 RTP 包；
7. 将 RTP 包进一步打成信令数据包；
8. 将此信令数据包通过信令通道发送到服务端。

需要注意的是，这里的第 3～6 步都是计算密集型的操作，为了避免阻塞 JS 主线程，需要将其用 WebWoker 包装，使这些步骤运行在非主线程中。同时，第 4 步视频编码操作需要借助 WebAsseamly 将第三方的 Video Codec（比如 OpenH264）编译为 Wasm 格式，导出为 JS 模块，方可被前端 App 引用。

视频流接收过程如下，每当从信令服务器接收到类型为 RTP 数据包的 PDU 时：

1. 将此 PDU 解析为 RTP 数据包；
2. 根据不同的 NALUnitType 将 RTP 数据包中的 RTP Payload 解析出来；
3. 将解析出来的 RTP Payload 发送到 OpenH264 模块进行解码，得到 YUV 格式的单帧图像；
4. 将单帧 YUV 图像转为 RGBA 格式的图像；
5. 通过 Canvas API 将 RGBA 格式的图像绘制到画布中。
   这里的第 3 步同样需要使用 WebWoker 包装，并引用基于 WebAsseamly 的 OpenH264 库，以避免 JS 主线程阻塞。 另一个需要注意的问题是，这里的第 4 步和第 5 步，需要使用 WebGL 进行性能优化，否则视频解码渲染的性能可能无法达到预期。

**这个方案完全移除了对 WebRTC 一揽子解决方案的依赖，并且初步实现了自己的媒体处理流水线。**这就意味着我们可以对视频媒体流进行进一步的定制化处理，从而支持更多面向客户的功能。

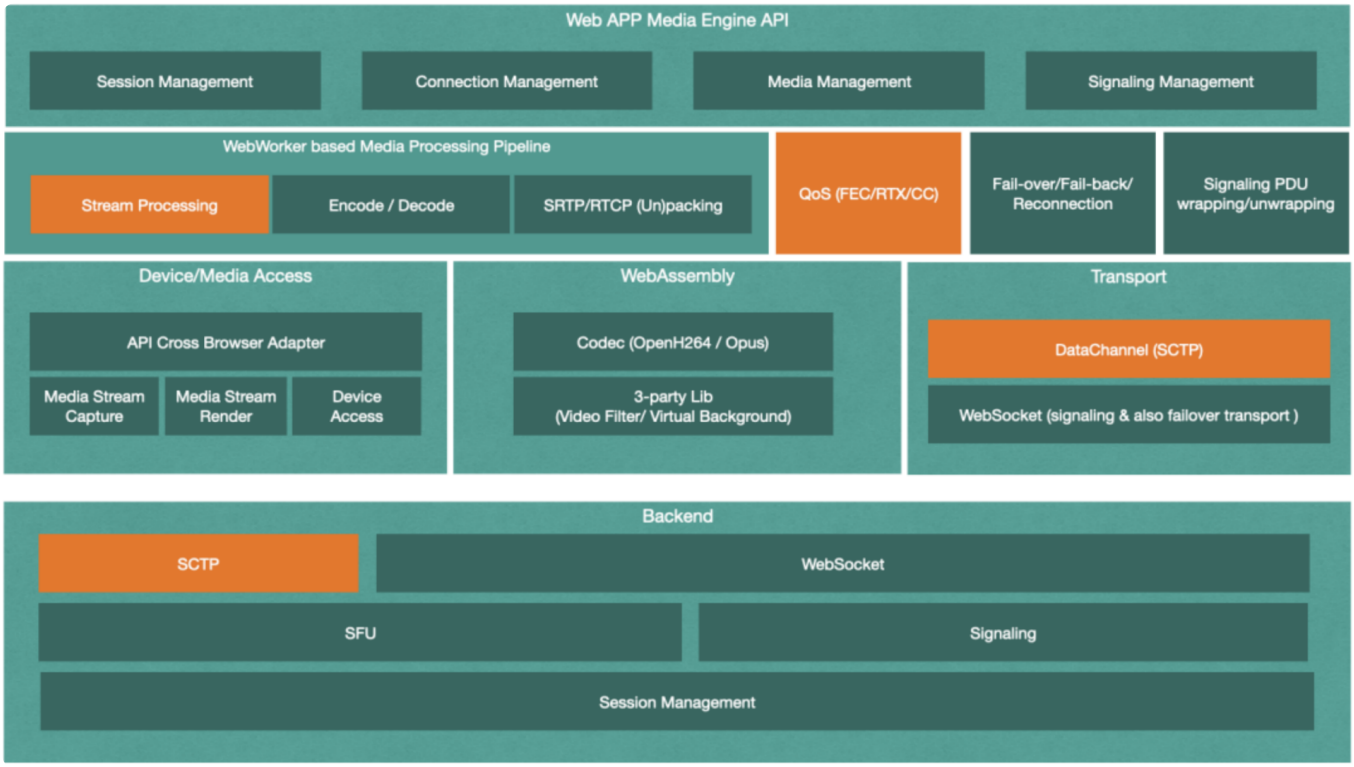
同时因为有了对媒体处理的完全掌控力，我们也可以在浏览器端实现自己的 QoS 方案，且这些方案都在 JS 层实现，可以兼容所有浏览器，从而避免服务端对不同浏览器 QoS 方案的支持。进一步说，将客户端的 QoS 实现策略通过 WebAsseamly 移植到 Web App 这边来，也是完全可行的方案。

但问题在于，作为失败转移方案，这里的媒体流数据是通过信令通道（WebSocket）进行传输的。在常规情况下，我们希望媒体数据能够走自己单独的数据通道，所以这里我们自然可以选择基于 SCTP（UDP based）传输协议的 DataChannel 作为媒体的传输通道。

## 优化方案

上面我们提到，当我们不再依赖 WebRTC 的一揽子解决方案，而是通过 WebAssembly Based OpenH264 Codec 构建自己的媒体处理流水线，并通过 WebSocket 进行传输时，我们一开始提到的无论是 QoS 的兼容性问题，还是客户端的媒体流定制需求，都可以得到满足，客户端还是服务端的架构都变得统一起来。

基于这种思路，我们可以继续在上面的失败转移方案的基础上，增加额外的基于 SCTP 传输协议的传输通道 DataChannel，以供媒体传输使用。此外，进一步丰富媒体处理流水线，使其能够实现滤镜、虚拟背景等媒体处理需求。最后，再加上统一的可跨浏览器的 QoS 算法，最终的架构图如下：




增加了如上图中所示的橘色模块以后，我们的系统开始变得丰满起来。**媒体定制处理**加入到了流水线，让我们能跨平台地满足更多的用户功能；完全自己实现的**跨浏览器 QoS 方案**，让前后端的算法变得统一起来，无需再考虑 QoS 的兼容性问题。

前端 DataChannel 和后端的 SCTP 的介入，让媒体流在绝大多数情况下，通过 DataChannel（SCTP）进行传输， 从而更好地利用 UDP 协议的优势，因为它的可靠性可以得到选择，从而弥补 TCP 按序重发机制在流媒体传输应用中的缺陷。而这里的 WebSocket，作为媒体数据传输通道，仅会发生在失败转移的特殊情况下。

看到这里，聪明的你一定发现了前端架构演进至此，也已经完全统一了，即正常流程和失败转移的流程除了传输层不同，其他模块完全共享，从而避免了多套解决方案需要同时维护的尴尬。

## 进一步优化

上述方案实现以后，我们将得到一个完整统一、基于纯浏览器且不受浏览器约束的视频会议系统。但这里是否真的没有任何问题，也没有任何优化空间了呢？

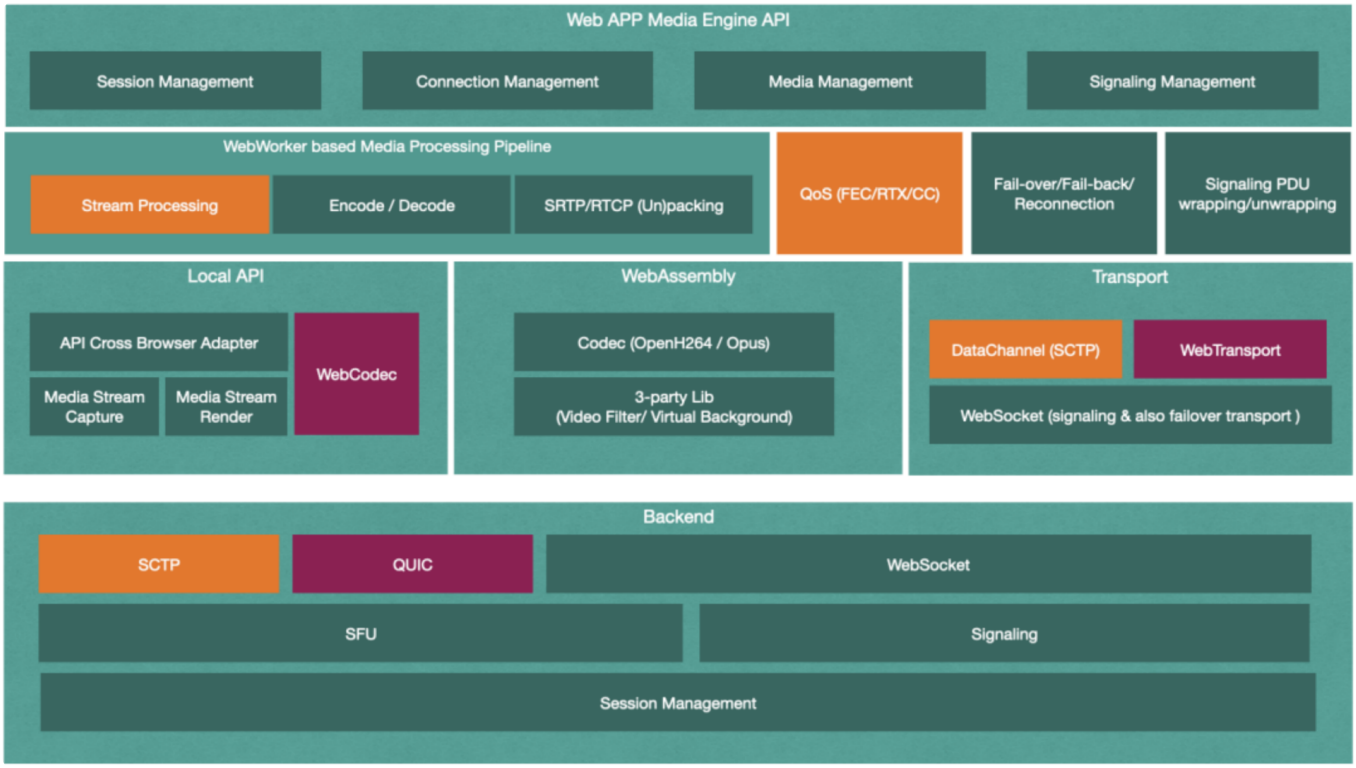
**答案显然是否定的。**

首先是基于 WebAsseamly 的 OpenH264 库的编码效率问题，尤其是在一些低端设备上，会导致 CPU 占用过高，并导致其发送或接收的视频发生卡顿。

其次就是 SCTP 的传输机制问题，虽然 SCTP 支持多流传输，能缓解队头阻塞的问题，但协议本身无法完全避免队头阻塞。除此之外，SCTP 协议建立连接时需要 4-RTT，其握手策略也可以继续优化，从而进一步减少视频首帧的加载时间。

Chrome 浏览器率先实现了基于 W3C 的 WebTransport 和 WebCodecs 规范。WebTransport 进一步优化了媒体传输通道，而 WebCodecs 规范则定义了浏览器所提供的更底层的媒体编解码接口。因而**我们可以直接调用浏览器提供的本地视频编解码器**，而不再需要调用 WebAssembly Based OpenH264 Codec 进行视频编解码，从而极大地提高了视频编解码效率。

因此，我们对前端架构进一步优化的架构图如下：




但需要注意的是，由于 WebTransport 和 WebCodecs 规范的提出时间较短，**目前只有 Chrome 浏览器支持**，因此这步优化也只能应用于 Chrome 浏览器，当然 Chromebook 用户也能享受到这两个规范所带来的福利。

以上就是这节课的全部内容，其中涉及到的很多技术栈这里只能点到即止，展开的话细节还有很多，感兴趣的同学可以通过文末的链接，获取相关技术的草案或者 RFC 做进一步了

解。

1. WebAssembly 工具链：https://emscripten.org
2. WebWorker：https://www.w3.org/TR/2021/NOTE-workers-20210128/
3. WebTransport：https://www.w3.org/TR/webtransport/
4. WebCodecs：https://www.w3.org/TR/webcodecs/
5. WebRTC（RTCDataChannel）：https://www.w3.org/TR/2015/WD-webrtc-20150210/
6. OpenH264 Codec：https://github.com/cisco/openh264/wiki
7. 拥塞控制（CC）相关协议：https://datatracker.ietf.org/wg/rmcat/documents/

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 14 | MP4 & FLV：不要再说AVI了

## 精选留言 (3)

写留言

**叫什么好呢**

2021-12-24

非常感谢章韦老师的分享。给老师点赞👍

**Geek_747837**

2021-12-24

就一讲时间还不如把一个点的说清楚些。一路听课到这里，就这讲感觉浪费时间了~